Training Graphs of Learning Modules for Sequential Data

NICOLAS CHAPADOS and YOSHUA BENGIO
University of Montreal

We consider the problem of the general composition of learning algorithms that must handle temporal learning tasks, in particular that of creating and efficiently updating the training sets in a sequential simulation framework. We start by enumerating the desiderata that composition primitives should satisfy, and underscore the difficulty of *rigorously* and *efficiently* reaching them. We then introduce a set of algorithms that accomplish the desired objectives. We conclude with a case-study of a real-world complex learning system for financial decision-making that uses the methods introduced herein.

Categories and Subject Descriptors: I.5.2 [**Pattern Recognition**]: Design Methodology

General Terms: Algorithms, Performance, Reliability

Additional Key Words and Phrases: Learning Algorithms, Sequential Validation, Economic and Financial Time-Series Forecasting

1. INTRODUCTION

Statistical learning algorithms [Hastie et al. 2001; Bishop 2006] have long found application in sequential decision-making tasks, particularly in economic and financial problems [Abu-Mostafa et al. 2001; Shadbolt and Taylor 2002]. In practical systems, individual learning algorithms are almost never used in isolation; rather, *complex networks* of modules[1] are designed that together provide the requisite functionality. For instance, in the context of financial portfolio management, complex investment strategies can result from the combination of learning-driven decisions, fixed decision rules (often used as safeguards), and both fixed and adaptive data preprocessing.

It is a challenge to rigorously and efficiently evaluate the performance of such "learning networks" (to be precisely defined below), especially with respect to criteria that take into account the whole sequence of decisions, such as financial perfor-

[1]Where each module can be a complete learning algorithm in the traditional sense, rather than, e.g. a single hidden unit in a neural network.

Authors' address: Nicolas Chapados and Yoshua Bengio, Computer Science and Operations Research Department, University of Montreal, C.P. 6128, succ. Centre-Ville, Montreal, Quebec, Canada, H3C 3J7.
Email addresses: {chapados,bengioy}@iro.umontreal.ca

mance measures (accounting for trading costs) in a portfolio management task. Due to the danger of *overfitting*, and associated *data snooping biases*, the vast majority of the machine learning literature (and more recently, the empirical finance literature[2]), provides, as a matter of course, out-of-sample evaluation of proposed models. Most of these take the form of a simple "train–test" split (also called an "estimation–validation" split), where an initial training set is used to fit model parameters, and an held-out test portion used to simulate post-deployment performance.

Unfortunately, for *sequential* decision-making tasks, single train–test splits exhibit several troublesome issues. First is the traditional question of *where to split*, with the objective of giving enough training data to train accurate models, and enough test data to derive low-variance performance estimates.[3]

The second and deeper problem is related to *inherent non-stationarities* in the data distribution: many tasks exhibit variables whose distribution change significantly through time. In macroeconomic forecasting, this may take the form of *structural breaks*, caused by, as observed by Clements and Hendry [1999, p. xxiii], evolving economies that "are subject to sudden shifts, precipitated by changes in legislation, economic policy, major discoveries, and political turmoil". Stock and Watson [1996] document the evidence of structural instability in 76 representative Post War US macroeconomic time series. In the finance literature, non-stationarities were reported for volatility in the form of (G)ARCH effects [Engle 1982; Bollerslev 1986; Campbell et al. 1997]; more recently, non-stationarities have also been recognized for expected returns and their effects on realized returns [Fama and French 2002]. It is obvious that ultimately, the whole question of non-stationarity hinges on the choice of model: the prism through which one analyzes the world may admit sufficient inherent complexity to explain phenomena for which a simpler apparatus would be misspecified. Nevertheless, our present goal is not to carry out an exhaustive overview of evidence in favour of non-stationarity in any particular domain, but to defensively assume that some of those effects can occur, and examine how one should deal with them.

The third problem is that single train–test performance evaluation occults the reality faced by a flesh-and-blood decision maker: no rational person would train a computer model on a limited subset of data and then let this model run for an arbitrary period of time—without as much as an update to the model even as new data becomes available. Rather, one would rationally want to quickly make use of all available information, promptly updating the model to reflect new data.

Sequential Validation [Gingras et al. 2002; Chapados and Bengio 2001] is an empirical testing procedure that aims to emulate the behavior of said rational decision maker. Inspired by the well-known technique of cross-validation [Stone 1974; Hastie

[2]Overfitting and data snooping biases were brought to the attention of the financial econometrics community by Lo and MacKinlay [1990]. White [2000] has proposed a "reality check" test to account for the effects of biases attributable to repeatedly using the same data to construct several models, although some studies have reported that the test lacks the power to distinguish between "good" and "bad" models in some important cases [Hansen and Lunde 2005].

[3]This point is not so inconsequential as appears at first glance: anecdotically, it occurred more than once to the authors to implement a model proposed in a paper that worked quite well on the train–test split used in the paper, but would fail disgracefully when tested on any other period: this can be seen as in instance of the so-called *dataset selection* bias.

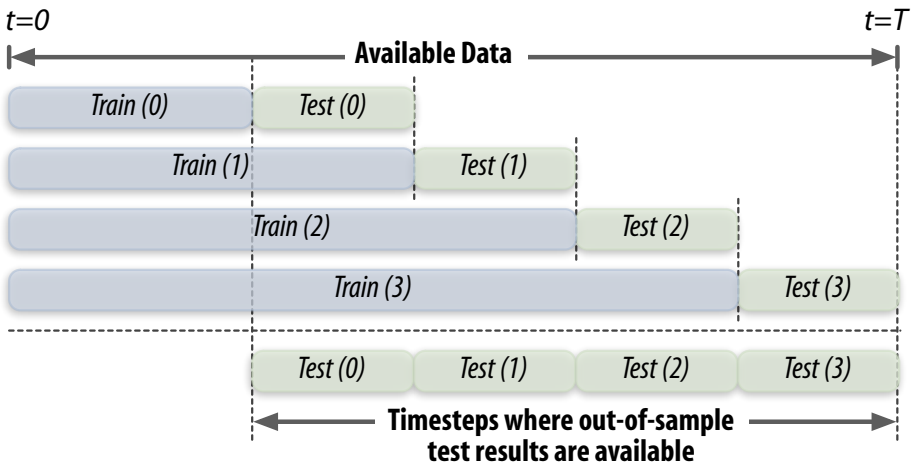


Fig. 1. Illustration of **sequential validation**. A model is retrained at regular intervals, each time tested on a small subset of data that immediately follows (in temporal order) the end of the training set. At each iteration, the test data from the previous iteration is added to the training set.

et al. 2001], its use is appropriate when the elements of a dataset cannot be permuted freely, such as is the case in sequential learning tasks. One can intuitively understand the procedure from the illustration in Fig. 1. One trains an initial model from a starting subset $Train(0)$ of the data,[4] which is tested out-of-sample on a data subset $Test(0)$ immediately following the end of the training set. This test set is then added to the training set for the next iteration, a new model is trained and tested on a subsequent test set, and so forth. As the figure shows, at the end of the procedure, one obtains out-of-sample test results for a large fraction of the original data set, all the while always testing with a model trained on "relatively recent data".

Stock and Watson [1999] use this technique in an economic forecasting context, calling it a "simulated out-of-sample forecasting methodology". They note that "this methodology provides a degree of protection against overfitting and detects model instability."

One sees immediately that, as opposed to a single "train–test" split, sequential validation is not only a very effective way of making the most of limited (temporal) data; it also provides an approach to deal with what we can call—informally—"slow" non-stationarities, i.e. progressive changes in the underlying generating process distribution.[5] It also applies, contrarily to cross-validation, to contexts where a single unbroken *sequence of decisions* must be maintained.

[4]"Starting" is meant in the temporal sense.

[5]If such non-stationarities are suspected, one should ensure that the training set remains of limited size—discarding old data—as the sequential validation proceeds, so as to at least ensure that distributionally different data eventually leaves the training set.

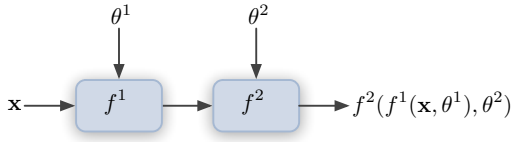


Fig. 2. Elementary composition of function approximators $f^1(\cdot,\theta_1)$ and $f^2(\cdot,\theta_2)$. The input vector is $\mathbf{x}$; θ_1 and θ_2 represent learned parameter vectors.

1.1 Challenges with Sequential Validation

As shown in Fig. 1, sequential validation entails repeated interleaved steps of training and testing. When dealing with complex *networks of composed learning algorithms*, the training step itself involves the creation of a training set for each adaptive element in the network. To see how this may arise, consider the very simple composition of two function approximators, $f^1(\cdot,\theta_1)$ and $f^2(\cdot,\theta_2)$, shown in Fig. 2. To unify notation, we shall denote by $\mathcal{T}_i^j$ and $\mathcal{U}_i^j$ respectively the *training* and *test* sets used for learner j at iteration i of sequential validation. Within a single iteration i of sequential validation, this network of two elements can be trained according to the following steps:

(1) Construct the training set $\mathcal{T}_i^1$ for f^1
(2) Train f^1 on $\mathcal{T}_i^1$ to obtain θ_i^1
(3) Construct the training set $\mathcal{T}_i^2$ for f^2 by computing the outputs of $f^1(\cdot,\theta_i^1)$ on the examples within $\mathcal{T}_i^1$; this is the **key compositional step** for training[6]
(4) Train f^2 on $\mathcal{T}_i^2$ to obtain θ_i^2

Across consecutive iterations of sequential validation, one can clearly anticipate some computational savings: for instance, the training set $\mathcal{T}_{i+1}^1$ has grown from $\mathcal{T}_i^1$ only by adjoining the elements in the corresponding iteration-i test $\mathcal{U}_i^1$; it does not need to be reconstructed from scratch. For $\mathcal{T}_{i+1}^2$, the situation is more unfortunate: in ordinary circumstances, no such shortcut exists for obtaining it from $\mathcal{T}_i^2$: since the behavior of $f^1(\cdot,\theta_i^1)$ must be assumed to differ arbitrarily from that of $f^1(\cdot,\theta_{i+1}^1)$ due to the difference in parameter vectors, the training set $\mathcal{T}_{i+1}^2$ would not generally be a strict expansion of $\mathcal{T}_i^2$—one needs to carry out step 3 in the above procedure in its entirety with the new θ_{i+1}^1 to construct $\mathcal{T}_{i+1}^2$. However, all is not lost: suppose that f^1 is non-adaptive and carries out fixed preprocessing; this implies that $\mathcal{T}_{i+1}^2$ becomes, in this case, a strict expansion of $\mathcal{T}_i^2$ and can be computed incrementally.

One might wonder why we appear to belabor this training-set incrementality issue. It turns out that for complex real-life networks constituted of a large number of arbitrarily-composed elements, the training-set construction steps may represent a significant fraction of the total running time of a simulation, particularly when frequent retrainings are requested in the sequential validation.[7] In addition, several

[6]Note that this step only provides the *input portion* for $\mathcal{T}_i^2$; the targets (desired outputs) need to be specified separately. In most applications of concern to us, this should not constitute a difficulty. For instance, if f^2 is to act as a predictor of asset returns, the target returns can be computed independently.

[7]This would arise when the learning algorithms themselves exhibit linear time complexity with respect to the length T of the training set; since training set construction is $O(T)$, its time complexity is on the same order of magnitude as the learning part *per se*.

learning algorithms can operate much more efficiently when given an "incremental" training set, such as the so-called *recursive estimators* for linear regrssion [Greene 2002].

Furthermore, in practical applications, one may wish to compose, not two or three elements, but several dozens—even hundreds—of them; where an illustration, depicted in Fig. 3, is further studied in §6/p. 23. In these cases, it becomes immensely time-consuming and error-prone for a human to identify by hand which elements can have their training set computed incrementally (from one iteration of sequential validation to the next), and which ones require full computation. To compound the difficulty, as we shall show below, it is sometimes not possible with economic time-series data to make this choice **statically** by just examining the graph of interconnections between elements; it may occur that the "status" of an element (namely, whether its training set admits incremental updates or must be reconstructed from scratch) changes along the simulation.

1.2 Goals and Organization of this Paper

Although most of the research in machine learning has concentrated on developing individual algorithms and establishing their theoretical properties, somewhat less attention has been paid to the engineering issues that surround the *systematic composition* of learning sytems.[8] In practice, as illustrated above, complex learning systems are composed of a large number of individual learning components; most often, the interconnections between those components is handled in an *ad hoc* fashion by the engineer, guided by the problem at hand. This approach tends to be laborious, brittle, and hard to scale.

This paper introduces a systematic procedure that is both correct and efficient for the creation of training sets when dealing with complex networks of processing elements, some of which are adaptive (i.e. learning algorithms), and whose performance is evaluated in a sequential validation framework, repeatedly interleaving training and testing.

The paper is organized as follows. In section 2, we explain specific issues in sequential learning (particularly when dealing with economic time-series data) that make out-of-sample performance evaluation somewhat of a challenge. In section 3 we introduce relevant notation as well as formally defining the learning and sequential-validation framework that we shall be assuming. Section 4 introduces general algorithms for the creation and update of training sets in a temporal simulation involving learning algorithms. We continue in section 5 with a domain analysis aimed at contrasting alternative architectures and underscoring the necessity of the proposed approach in order to realize all desired composition scenarios. We conclude (section 6) with a case study of a practical deployment of the algorithms herein introduced in the context of investment fund management.

[8] These issues are quite different from those tackled by meta-algorithms such as bagging [Breiman 1994] and boosting [Freund and Schapire 1996], which are concerned with specific compositional forms to derive *new learning algorithms* with precise properties. Our concerns are somewhat related to those faced by stacking [Wolpert 1992]; we discuss this point in the proceeds.

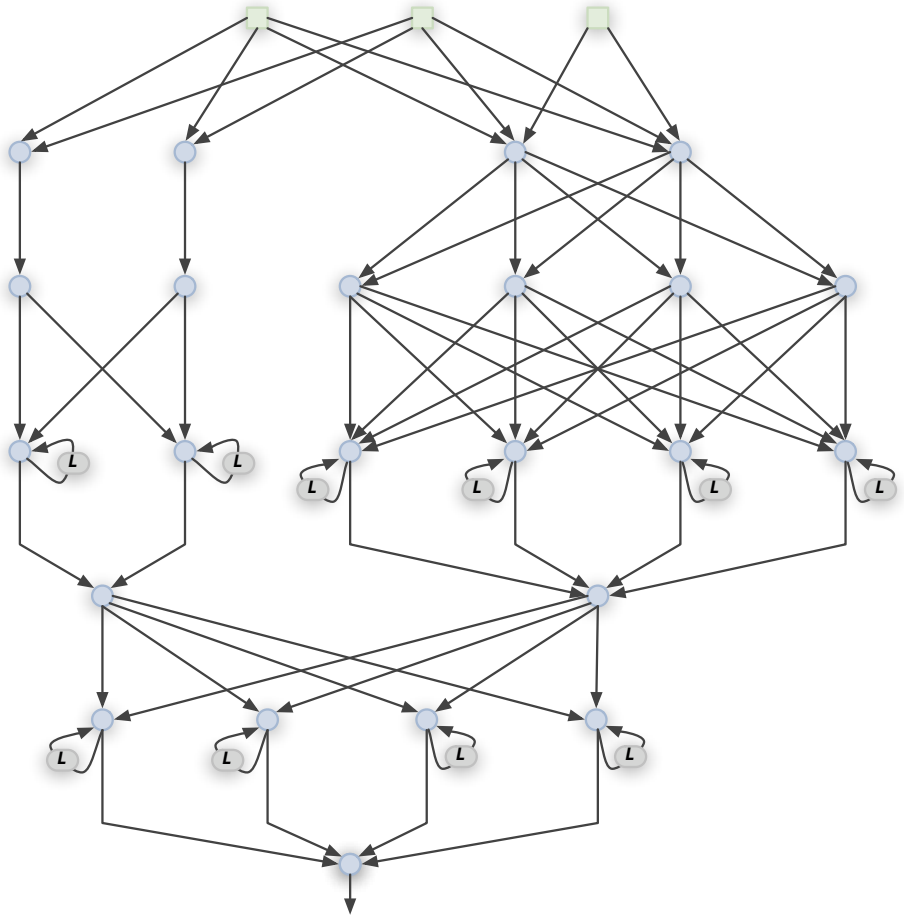


Fig. 3. Example of complex composition of learning algorithms required by a financial decision-making application (outlined in §6/p. 23) The top row represents external inputs and each lower-level circle represents a complete learning algorithm; the arrows represent the functional data flow. The loops labeled with an L represent time-lagged connections.

2. CORRECTNESS ISSUES IN SEQUENTIAL LEARNING

Since our main focus is on financial decision-making, we are seeking an experimental framework that is as close as possible to the situation faced by an actual economic agent who must make decisions in real time. In particular, even though typical simulations involve the use of large amounts of historical data, we rely on an experimental methodology that seeks to avoid *data snooping biases* [Lo and MacKinlay 1990; White 2000] to the greatest possible extent.

One simple but common cause of error when evaluating forecasting or decision models within a standard out-of-sample setting is to mistakenly use *data from the future* when constructing a model for a given day (assuming an ongoing simulation). This tends to yield an optimistically-biased estimate of future performance, since

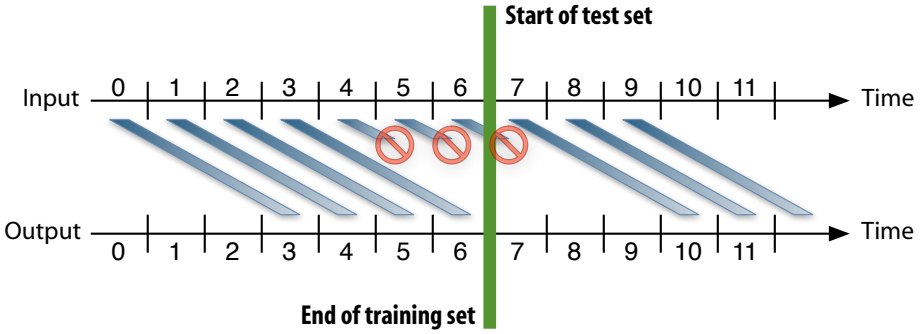


Fig. 4. Illustration of forecasting at horizon h; in a train–test setting, the last h observations in the training set must be discarded since their corresponding target lies "in the future" (in the test set).

the knowledge of future outcomes obviously is of great help when trying to make a forecast of those outcomes.

This situation is illustrated in Fig. 4, wherein forecasting at horizon $h = 3$ is attempted. Because of the horizon, we are capable of evaluating the quality of a decision made at time t only at time $t+3$ (on the figure). At the boundary between the training and test sets, we have h training examples for which we are never "legally" capable of measuring performance, since the target information is part of the test set. These elements must be discarded from the training set in order to arrive at a *causal* estimation of performance, for otherwise targets for the last h examples in the training set would overlap with the test set, yielding the optimistic performance estimation bias.

2.1 Economic Data

Causality implies that we treat macroeconomic data with particular care when performing historical simulations. Clements and Hendry [1999] note that such data series "may be inaccurate, prone to revision, and are often provided after a non-negligible delay." Contrarily to an oversimplified textbook picture where macroeconomic data series are provided and static, the reality is cloudier: most series[9] are updated *post hoc* through a process of successive revisions. Upon releasing a new value for most of the significant economic indicators, the past value (generally that of the previous month) is revised as well. It may happen that revisions go further in the past, as the example of Fig. 5 illustrates.

It is unfortunate that most data providers only provide the **last published revision** as the "true value" of a series; this obscures the real nature of the impact of economic news releases on the financial markets. To understand this impact, it is useful to recall that most economic variables of importance are tracked by a number of analysts who make forecasts as to their outcome; the average value of these forecasts is called the *consensus estimate* for a variable. When new data is

[9]At least in the United States and Canada.

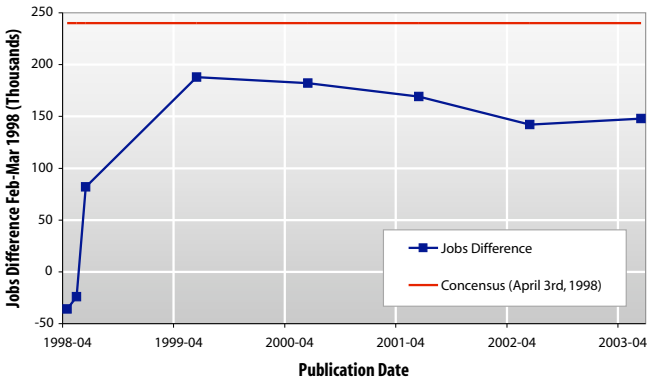


Fig. 5. Example of *post hoc* revisions to macroeconomic time-series. U.S. non-farm payroll difference between February and March 1998, first published on April 4th 1998, along with the most recent concensus estimate before the release date (on April 3rd 1998). The plot shows that significant revisions are made to the data up to several years after its initial release. Data provided by the Federal Reserve Bank of St. Louis.

released regarding a variable, it is often argued by practitioners that the following quantities have the greatest market impact:[10]

—The difference between the actual value and the most recent consensus estimate (the so-called *surprise*).

—The difference between the actual value and the last available revision of the previous time-step (e.g. month) value.

Some academic studies lend credibility to this view, particularly regarding its consequences on market volatility [Hautsch and Hess 2002; Hess 2001]. This "market reaction hypothesis" implies that revisions to a series value are, to a first approximation, ignored by the market.[11] It also suggests that carrying out simulations using the commonly-available historical macroeconomic data may introduce hard-to-quantify noise, especially when the revised values are of a different sign than the originally-published values, when compared against the consensus estimates.[12] In all cases, when one cares about the **causality of simulations**, it is incorrect to use such revised data: the revisions (namely, the most recent series values) would not have been available to a decision-maker acting in real time, thence should not be used.

Luckily, it is now becoming possible to have access to so-called "vintage data," containing the history of all revisions released about a series.[13] Availability of this

[10]In the sense of introducing local volatility.

[11]To a large extent, this can be explained by the fact that when a revision is issued at a later date, its impact is submerged by that of the most recent series values. For instance, if a revision is issued in May for the March employment, its impact is overshadowed by the first release of the April employment figures.

[12]It further suggests that models that use macroeconomic variables as inputs should include consensus estimates as well.

[13]For instance, in the United States, the Federal Reserve Bank of St. Louis now makes vintage

data enables us to conceive of historically-accurate simulations of decision-making processes. Nevertheless, practical complications quickly come to light, especially in areas of database management and learning algorithms.

2.2 Vintage Data and Database Management Issues

The most obvious is related to database management issues: a "vintage series" can no longer be considered a single time series, but really is a **collection** of such series, one for the view of the past that was in effect on each given day.[14] One common manner of handling this data is through a *two-date* convention, wherein all financial time-series observations are recorded using two dates instead of one:

—The **observation date** records the nominal date for which the observation is about. For example, the unemployment rate for September 2007 would be recorded with an observation date of (say) 30 September 2007, regardless of the date at which revisions are published.

—The **publication date** records the moment at which the information becomes available. For historical time-series data, this date comes after (or precisely on) the observation date.

To give a concrete example, suppose that the unemployment rate for September 2007 is first released on 15 October 2007, then subject to two monthly revisions. The observation and publication dates would be recorded as follows in the database:

Observation Date	Publication Date
30 September 2007	15 October 2007
30 September 2007	15 November 2007
30 September 2007	15 December 2007

2.3 Vintage Data and Learning Algorithm Issues

The introduction of *post hoc* revisions to time-series data presents a further obstacle in the context of sequential validation: when computing, for instance, the difference between a variable and its consensus, we want this difference to be taken using **first-published** series values—which remain constant throughout the simulation. In contrast, when using a variable as a *level* (also known as *stock variables*, we rather want to use **most-recently-published** (at the time of making the simulated decision) series values. The crucial difference is that, in the latter case, the "view of the past" regarding a series may change: this occurs when the simulation crosses a date where a new revision was published about a previous observation date.

In particular, training sets that could have been constructed incrementally in a sequential validation framework (§1.1/p. 4) may have to be mended more deeply to accomodate history revisions that could affect some series. This might require relinquishing incrementality of training-set updates in situations where it would otherwise have been feasible. Additionally, opportunities for incrementality now becomes *data-dependent*: contingent on whether and when a series is revised, training sets that depend on that series may be able to benefit from incremental updates.

---

series available; they are what allowed us to plot Fig. 5 the first place.

[14] Although we focus on time series of historical data, the same constructions could manifestly be conceived for series of forecasts of future events.

Still, context-specific realities and optimizations mitigate this unpropitious outlook. In particular, because this process of history revision affects only macroeconomic series, which are generally published monthly, we maintain hopes of preserving a certain level of computational efficiency (and we assume that most reasonable simulations with financial data are at a daily or higher frequency). Furthermore, higher-frequency data (including prices) remain largely unaffected by this phenomenon.[15] Finally, we introduce below an optimization that efficiently deals with the scenario of revisions that are made in the last simulated time-step.

3. LEARNING FRAMEWORK AND NOTATION

We now formalize the learning concepts that we shall be using. We assume a discrete-time framework, $t \in \mathbb{N}$, where a single *time period* (e.g. a day or a month) elapses between times t and $t+1$. We further assume that T time steps of data are available, numbered from 0 to $T-1$.

3.1 Learning

We define learning primitives that are useful for controlling *dynamical systems*, wherein a temporal dimension is significant. Sequential validation, as introduced below, makes use of those primitives. Operationally, as depicted in Fig. 6, we can view the learning system as making a decision at each time-step as the result of an *output computation function* that yields an output y_t given a current input x_t, state ξ_t and parameter vector $\boldsymbol{\theta}$ (explained next). The *state* is a summary of the decisions that have been made in the previous time-steps and that are relevant for making the current decision. For example, the state may specify the location of an agent in a gridworld, or the currently-held assets in a portfolio management task. More generally, the state constrains the set of admissible outputs (actions) in the current time-step.[16] The state evolves according to a *state-transition function*, accepting the current output y_t in addition to the same inputs as the output computation function. A particular sequence $\langle x_t, y_t, \xi_t \rangle_{t=0}^{T-1}$ of inputs-outputs-states is called a *trajectory*.

Finally, the learner's behavior is assumed to be governed by a set of adjustable parameters that are contained in the parameter vector $\boldsymbol{\theta}$. These are obtained as the result of a *batch training function*, which yields $\boldsymbol{\theta}$ given a training set. Note that training, in this framework, is independent of any given trajectory and can be thought of as operating on a different level: it is both possible for the same parameters to be applied to many different trajectories (for instance, in a Monte Carlo simulation context), or for the parameters to change in the middle of a single trajectory (as illustrated in Fig. 6). The latter would typically occur in a sequential validation scenario.

The following definition makes those concepts more precise.

DEFINITION 1. *A **temporal learning algorithm** is a quadruple* $\langle f, g, h, \xi_0 \rangle$ *where*

[15] Although, as this is written in 2008, some providers of commodity futures data only make daily trading volume and open interest available with a 24-hour delay.

[16] It is sometimes assumed that the state is a finite-dimensional vector; we shall need no such assumption about the representation of the state object in the current definition.

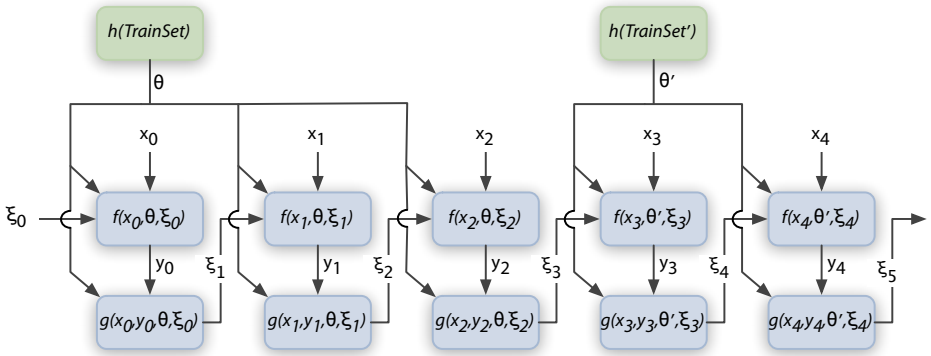


Fig. 6. Illustration of the framework that is assumed for temporal learning. As outlined in the text, $f(x, \boldsymbol{\theta}, \xi)$ is an output-computation function, given current inputs x, state ξ and learned parameters $\boldsymbol{\theta}$; $g(x, y, \theta, \xi)$ is the state-transition function, and $h(\mathit{TrainSet}) \rightsquigarrow \boldsymbol{\theta}$ carries out time-independent batch training, yielding the parameter vector. Also shown is the fact that batch training can be arbitrarily interleaved with output computations and state transitions (h called with a new training set, $\mathit{TrainSet}'$).

Output Computation. $f : \mathcal{X} \times \Theta \times \Xi \mapsto \mathcal{Y}$ *is a function that given a set of inputs* $\mathbf{x} \in \mathcal{X}$*, a set of parameters* $\boldsymbol{\theta} \in \Theta$ *and a current state* $\xi_t \in \Xi$ *computes a set of outputs* $\mathbf{y} \in \mathcal{Y}$.[17]

State Transition. $g : \mathcal{X} \times \mathcal{Y} \times \Theta \times \Xi \mapsto \Xi$ *is a function that given a set of inputs* $\mathbf{x} \in \mathcal{X}$*, previously-computed outputs* $\mathbf{y} \in \mathcal{Y}$*, a set of parameters* $\boldsymbol{\theta} \in \Theta$ *and a state* $\xi_t \in \Xi$ *at time* $t \geq 0$ *computes a state for the next time-step* ξ_{t+1}*. State transitions are deterministic with respect to* ξ *(which may represent a probability distribution, as we shall see below).*[18]

Training. $h : \mathcal{X}^\ell \times \mathcal{Y}^\ell \mapsto \mathcal{B} \times \Theta$ *is a function that given* ℓ *input-target pairs returns a set of parameters* $\boldsymbol{\theta} \in \Theta$ *as well as an indicator* NON-TRIVIAL $\in \mathcal{B} = \{\text{TRUE}, \text{FALSE}\}$ *if the training has been "non-trivial", namely that functions* f *and* g *can be assumed to behave differently with the new set of parameters.*

Initial State. $\xi_0 \in \Xi$ *is an initial state from which output computation and state transition operations can be started.*

Note that we assume that an *initial training* is performed before the output computation and state transition functions can be used for the first time.

[17]The precise spaces in which these quantities lie is unimportant, but for our purposes it will be enough to assume they are *sets of named vectors of reals*, for instance $\mathcal{X} \ni x = \{\langle name_i, x_i \rangle\}$ with $name_i$ some learner-dependent identifiers and $x_i \in \mathbb{R}^{n_i}$ for some integer n_i. This representation is both more powerful and convenient than the traditional fixed-dimensional vector provided as input to a learning algorithm, for the input to a learner can generically be specified as the set union of the outputs of the learners on which it depends, without necessitating the introduction of arbitrary concatenation rules.

[18]For notational simplicity, we assume that the state space Ξ is time-invariant, but this is not strictly required for the rest of the exposition.

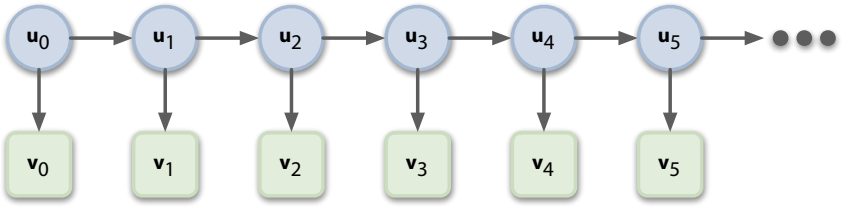


Fig. 7. Generative view of the Kalman filter as a graphical model.

3.1.1 *Example: The Kalman Filter.* To illustrate the generality of the above framework, we briefly consider how the well-known Kalman filter ([Kalman 1960]; see, e.g., Kailath et al. [2000] for an introduction) can be expressed in terms of the above primitives. A probabilistic graphical model view of the dynamical system assumed by the filter is given in Fig. 7 [Roweis and Ghahramani 1999]. The (uncontrolled) dynamics of the system are

$$\mathbf{u}_t = \mathbf{A}_t \mathbf{u}_{t-1} + \mathbf{w}_t, \qquad t > 0 \tag{1}$$

$$\mathbf{v}_t = \mathbf{H}_t \mathbf{u}_t + \mathbf{z}_t, \qquad t \geq 0 \tag{2}$$

with

$$\mathbf{w}_t \overset{\text{iid}}{\sim} \mathcal{N}(0, \mathbf{Q}_t), \qquad \mathbf{z}_t \overset{\text{iid}}{\sim} \mathcal{N}(0, \mathbf{R}_t), \qquad \mathbf{u}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Gamma}_0)$$

and $\mathbf{u}_t, \mathbf{w}_t \in \mathbb{R}^m, \mathbf{A}_t, \mathbf{Q}_t \in \mathbb{R}^{m \times m}, \mathbf{v}_t, \mathbf{z}_t \in \mathbb{R}^n$, $\mathbf{H}_t \in \mathbb{R}^{n \times m}$ and $\mathbf{R}_t \in \mathbb{R}^{n \times n}$. The notation $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ represents a normally-distributed random variable with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$.

In this system, $\mathbf{u}_t$ represents the *state* at time t and is assumed to be *hidden* (unobserved).[19] It must be inferred from a sequence of observations (measurements) $\mathbf{v}_t$. From the initial conditions and system equations, and assuming known values for $\mathbf{A}_t, \mathbf{Q}_t, \mathbf{H}_t$ and $\mathbf{R}_t$, Kalman derived recursive equations for computing the distribution of the sequence of hidden states given a sequence of observations. The distribution of the initial state $\mathbf{u}_0$ is assumed to be normal with given mean $\boldsymbol{\mu}_0$ and covariance matrix $\boldsymbol{\Gamma}_0$.

We shall be concerned with the **filtering** (or tracking) view, wherein given an observation sequence that extends until current time t and a state distribution estimate for $\mathbf{u}_{t-1}$ at time $t-1$ we seek to infer the state distribution at time t.[20]

To express the Kalman filter in the temporal learning framework, we shall pose the following equivalencies:

[19] It should be noted that "state" is used with a different meaning in discussing the Kalman filter than is used in def. 1: in the first case, the state is a single vector $\in \mathbb{R}^m$ (and follows a normal distribution under the Kalman filter assumptions); in contrast, the notion of state for def. 1 is an abstract object that encompasses arbitrary sufficient statistics about the past. In the present section, the use of "state" by itself refers to the Kalman state; when talking about state in the sense of def. 1, we shall use "learner state".

[20] This is different from the **smoothing view** wherein one is given *a priori* the complete observation sequence from 0 to T, from which one may infer the most likely state trajectory. However, smoothing is an acausal operation that requires knowing about the future, which we shall not further consider.

—The temporal-learning **set of parameters** $\boldsymbol{\theta}$ is made up of the Kalman state-transition matrices, observation matrices, and initial state distribution, respectively $\mathbf{A}$, $\mathbf{Q}$, $\mathbf{H}$, $\mathbf{R}$, $\boldsymbol{\mu}_0$ and $\boldsymbol{\Gamma}_0$. Note that although the transition and observation equations (1)–(2) support time-varying parameters, we shall assume that they do not vary in order to make learning tractable.

—The **learner state** at time t is made up of the *predicted* mean and covariance matrix of the Kalman state for time t given information until time $t-1$, respectively written as $\hat{\boldsymbol{\mu}}_{t|t-1}$ and $\hat{\boldsymbol{\Gamma}}_{t|t-1}$. Since eq. (1) and (2) are linear and the initial Kalman state $\mathbf{u}_0$ is assumed to be normally-distributed, the Kalman state for all subsequent time-steps will also be normally-distributed, and it is sufficient to keep track of only the current mean and covariance matrix. This implies that the *learner state* consists of just these two quantities.

—The temporal-learning **input** is an observation vector $\mathbf{v}_t$ provided to the Kalman filter. Note that what is termed "input" here is really an "output" of the generative model (2); however, from the learner standpoint, these observations are the inputs required for inference. The Kalman filter can also be augmented to cover the *controlled case*, wherein an exogenous control variable is used to alter the state dynamics (1); in this case, the control variable would also be provided as part of the learner inputs.

—The temporal-learning **output** is a *corrected* mean and covariance matrix of the Kalman state, that incorporates the information given by the observation vector $\mathbf{v}_t$. These are written as $\hat{\boldsymbol{\mu}}_{t|t}$ and $\hat{\boldsymbol{\Gamma}}_{t|t}$.

From these equivalencies, the temporal-learning operations are defined as follows:

Initial State. Consists of $\boldsymbol{\mu}_0$ and $\boldsymbol{\Gamma}_0$ obtained by a previous call to the training procedure.

Output Computation. Correct the state $\hat{\boldsymbol{\mu}}_{t|t-1}$ and $\hat{\boldsymbol{\Gamma}}_{t|t-1}$ to take into account the time-t observation $\mathbf{v}_t$. Start by computing the residual between the observation and its expected value under the predictive distribution

$$\hat{\mathbf{z}}_t = \mathbf{v}_t - \mathbf{H}\hat{\boldsymbol{\mu}}_{t|t-1}$$

whose covariance matrix is given by

$$\mathbf{S}_t = \mathbf{H}\hat{\boldsymbol{\Gamma}}_{t|t-1}\mathbf{H}' + \mathbf{R}$$

yielding the so-called *Kalman gain* factor

$$\mathbf{K}_t = \hat{\boldsymbol{\Gamma}}_{t|t-1}\mathbf{H}\mathbf{S}_t^{-1}.$$

The updated estimators for the state mean and covariance matrix are seen as *corrections* to the previous estimators, and given by

$$\hat{\boldsymbol{\mu}}_{t|t} = \hat{\boldsymbol{\mu}}_{t|t-1} + \mathbf{K}_t\hat{\mathbf{z}}_t$$

and

$$\hat{\boldsymbol{\Gamma}}_{t|t} = (I - \mathbf{K}_t\mathbf{H})\hat{\boldsymbol{\Gamma}}_{t|t-1},$$

with I the identity matrix. The last two quantities constitute the output of the learner at time t.

State Transition. Compute the next state as a linear forecast implied from the state-transition equation (1), the result of the current output (corrected mean $\hat{\boldsymbol{\mu}}_{t|t}$ and covariance matrix $\hat{\boldsymbol{\Gamma}}_{t|t}$),

$$\hat{\boldsymbol{\mu}}_{t+1|t} = \mathbf{A}\hat{\boldsymbol{\mu}}_{t|t} \tag{3}$$

$$\hat{\boldsymbol{\Gamma}}_{t+1|t} = \mathbf{A}\boldsymbol{\Gamma}_{t|t}\mathbf{A}' + \mathbf{Q} \tag{4}$$

Training. Model parameters $\mathbf{A}$, $\mathbf{Q}$, $\mathbf{H}$, $\mathbf{R}$, $\boldsymbol{\mu}_0$ and $\boldsymbol{\Gamma}_0$ can be estimated by the Expectation-Maximization algorithm [Dempster et al. 1977; Neal and Hinton 1998] to maximize the likelihood of sequences of observations within a training set. Derivations of the EM reestimation equations specific to the Kalman filter are given by Shumway and Stoffer [1982] and Ghahramani and Hinton [1996].

3.1.2 *Additional Remarks.* Some additional remarks are in order regarding the definition of a temporal learning algorithm:

▶ Training versus State Variables : Training does not involve any state variable; in other words, training, as defined above, occurs in "batch mode", independently of any current test trajectory. Philosophically, we can consider that training defines the identity of a learner (in general), whereas states govern its behavior in a particular context. The prototypical example of such a learner is the previously-introduced Kalman Filter, wherein transition matrices are estimated by maximum likehood from a large historical dataset during training, but filtering for a particular trajectory is performed by the combination of output-computation and state-transition functions.[21]

▶ Pure Preprocessing or Online Learning : We allow Θ to be the empty set $\emptyset$. This is useful to allow *non-adaptive* elements in a temporal learning network, namely fixed processing elements or strictly online learning modules [Saad 1999], for which all learned parameters occur as state variables. This special case is recognized and optimized for by the algorithms introduced in §4/p. 15.

▶ Input Data Representation : Raw data can be represented as learners that do not take any inputs,[22] for which $\Theta = \emptyset$, and that output "constant" results that depend only on the current time step. History revisions can be effected by returning the indicator NON-TRIVIAL = TRUE from the training function h when such revisions occur; this notifies dependent learners that incrementality assumptions no longer hold on their training sets. Note that in the current framework, history revisions are relevant only during training time, which operates on the entire data history. (In contrast, output computation time operates only on the incremental information available for the new time-step.)

▶ Bayesian Estimation : The training function h can also encompass the Bayesian case, wherein one does not estimate a single vector of parameters, but a *posterior*

[21] Also note that the separation between training and output computation/state transition implies that the *interpretation* of the state variables is independent from the results of training.

[22] Besides the current and previous "end-of-training" dates, which can be viewed as meta-inputs. As explained below, this lets the learner know that a history revision boundary has been crossed in the sequential validation.

probability distribution (given the training data) over the space of parameters. This amounts to having a $\boldsymbol{\theta}$ with a more complex structure, which our notation does not prohibit. We would also assume that the prior distribution over parameters is passed along with the training set. However, for simplicity, the rest of this paper assumes the point-estimation case.

3.2 Temporal Learning Networks

As stressed in the introduction, typical learning algorithms do not occur in isolation; we now define the graph structure that enables their composition.

DEFINITION 2. *A **temporal learning network** is a directed graph $G = \langle \mathcal{V}, \mathcal{E} \rangle$ where*

Vertices. $\mathcal{V} = \{v^j\}$ is a set of J temporal learners (elements that satisfy def. 1) $v^j = \langle f^j, g^j, h^j, \xi_0^j \rangle, j = 1, \ldots, J$;

Edges. $\mathcal{E}$ is a set of triples $\langle i, j, k \rangle$, where $v^i, v^j \in \mathcal{V}$ represent a directed link between learners v^i (source) and v^j (destination), and $k \in \mathbb{N}$ is a temporal lag on the connection, as explained below.

Furthermore, we let $\mathcal{E}_0 \stackrel{\triangle}{=} \{\langle u, v, k \rangle \in \mathcal{E} : k = 0\}$, the set of lag-0 edges. The subgraph $G_0 = \langle \mathcal{V}, \mathcal{E}_0 \rangle$ must be acyclic. We assume, without loss of generality, that the learners are numbered to satisfy the topological ordering, such that $\langle i, j, 0 \rangle \in \mathcal{E}_0 \Longrightarrow i < j$.

The intent of this definition is to capture output–input connections between learners. A lag-0 edge $\langle u, v, 0 \rangle$ represents the basic building block of functional composition: we assume that the output computed by learner u at time t serves as input to learner v at the same time-step.[23] Likewise, a lag-k edge $\langle u, v, k \rangle, k > 0$ represents a delayed connection: the output of u at time $t - k$ is used as the input of v at time t. Clearly, in order to have well-defined propagation at time t, we require the subgraph of lag-0 edges $\mathcal{E}_0$ to be acyclic such that an ordering exists in which to perform the output-computation operations; this can be found by an elementary topological sort algorithm [Cormen et al. 2001]. No such ordering is required for higher-order lags. We will assume that a sentinel "missing value" $\perp$ is returned whenever a value for a time-step $t < 0$ is required.

4. ALGORITHMS

In this section, we present a set of algorithms to evaluate the performance of a temporal learning network within the sequential validation framework, as well as efficiently updating the training set of each learner.

The algorithms are introduced in several logical parts. First, the overall driver of the simulation is the SequentialValidation procedure, which does not present any technical difficulty. We then present the training-related procedures (Train and UseOnTrain) and the output-computation and state-transition ones (ComputeOutput and TransitionState). The latter procedures act as "wrappers" around the primitive

[23] If learner v receives multiple inputs, it is provided with the union of inputs coming from all sources.

Table I. Global variables assumed to persist between procedure calls

Variable	Procedure(s)	Meaning
$\hat{\xi}_t^j$	SequentialValidation, ComputeOutput, TransitionState	Learner j state variable at time t for the test trajectory
$\tilde{\xi}_t^j$	SequentialValidation, Train, UseOnTrain	Learner j state variable at time t for the train trajectory
$\boldsymbol{\theta}^j$	Train, UseOnTrain, ComputeOutput, TransitionState	Learner j current parameters
X^j	Train, UseOnTrain	Learner j current training set
Y^j	Train, UseOnTrain	Learner j outputs computed on the current training set
x_t^j	ComputeOutput, TransitionState	Learner j input variables at time t
y_t^j	ComputeOutput, TransitionState	Learner j output variables at time t, for $t \geq 0$; "missing value" $\perp$ for $t < 0$

functions f^j, g^j and h^j for each learner j, introduced in §3.1/p. 10. For simplicity of exposition, we assume a strict sequential validation framework, in particular that we have a single time history which is used for creating both the training and test sets.

4.1 Variables and Notation

Table I lists the global variables that are assumed to persist between procedure calls. In a concrete implementation, it is straightforward to convert those to an "object-oriented" arrangement.

Regarding notation, we assume we can take a time-t "slice" of a training set X^j by writing X_t^j, which produces an element suitable for using as input to the output-computation function f^j. We further assume that we can row-wise construct such a training set by assigning to each time-t slice separately, e.g. $Y_i \longleftarrow f(\ldots)$.

4.2 Sequential Validation

The SequentialValidation procedure starts (lines 3 and 4) by initializing the state variables corresponding to a **train** and a **test** trajectory. The need for two separate trajectories arises from the necessity to call the output-computation functions f^j from two separate contexts: first, during the normal test step of sequential validation (see Fig. 1) to compute outputs on *test inputs*, and second after training learner j to compute the outputs on *train inputs* (i.e. on the elements of the training sets), which is necessary for constructing the training set of any dependent learner. As will be made clear below in the description of Train and UseOnTrain, it is frequently necessary to reinitialize the training trajectory to time-step 0, but the test trajectory should never be reinitialized during the execution of sequential validation, since this would correspond to rewriting history (and would be an action unavailable to a decision-maker).

Within the main loop of SequentialValidation (lines 8–13), output computation (testing) occurs at every time-step while training is carried out according to a

completely-independent, asynchronous, schedule passed as an argument to the procedure. The *training-schedule* is simply the set of time-steps on which training is allowed to occur, where we assume that $0 \in$ *training-schedule*, so that outputs are never computed on an untrained learner. At time t, the outputs are always computed given the most recent parameter estimates, resulting from training at time $t_{previous}$.[24]

Listing 1. Sequential Validation

```
1  def SequentialValidation(training-schedule):
2      # Initialize test (ξ̂) and train (ξ̃) trajectories
3      ξ̂_0^j ⟵ ξ_0^j,   j = 1, ..., J
4      ξ̃_0^j ⟵ ξ_0^j,   j = 1, ..., J
5      t_previous ⟵ −1
6
7      # Interleave training and testing
8      for t in 0, ..., T − 1:
9          if t ∈ training-schedule:
10             Train(t, t_previous)
11             t_previous ⟵ t
12         ComputeOutput(t)
13         TransitionState(t)
```

4.3 Training

The Train procedure wraps individual learners' training functions h^j and keeps track of a "dirtiness" status *is-dirty*^{j} for each learner. This status determines whether the associated UseOnTrain procedure is allowed to compute the learner's outputs on the training set incrementally or not. Train starts by considering each learner "clean" (lines 2–3). Then, in the order given by the topological sort of the lag-0 edges in $\mathcal{E}$, it constructs the training set for learner j from the outputs on the training set of learners connected to j through a lag-0 edge (line 7). For simplicity of notation, this Train pseudocode does not handle delayed connections at the training level. If training sets can be viewed as matrices, then this is easily handled through shifting the inputs matrices Y^i by an appropriate number of rows and introducing rows of missing values as appropriate.

The results of the learner's training function h^j are used in two ways: First, the indicator of whether the training was non-trivial is used to mark the learner dirty if it was not already; a "dirty learner" then propagates its filth to its train-time dependents (lines 13–15). This process is illustrated in Fig. 8. Finally, the

[24] With this arrangement, care must be taken when the learner parameters can be used to introduce a scale transformation on some variables, which in turn are used as *lagged inputs* to other learners. For example, suppose that learner 1 computes the principal components (PCA) of a set of input variables, whose first-differences $\mathrm{PCA}_t - \mathrm{PCA}_{t-1}$ are used as inputs by learner 2. Assume that a retraining occurs at time τ. If lagging is implemented "naïvely" by buffering the previous-time outputs, at time τ the PCA difference would use PCA_t computed with time-τ parameters, but PCA_{t-1} computed in an altogether different space with time-$\tilde{\tau}$ parameters, where $\tilde{\tau}$ is the time of the previous training.

new parameters $\boldsymbol{\theta}^j$ are used to compute the learner's outputs on the training set through UseOnTrain.

Listing 2. Training

```
1  def Train(t, t_previous):
2      for j in 1,...,J:
3          is-dirty^j ⟵ FALSE # Initially, everybody is clean
4
5      for j in 1,...,J:
6          # Create training set X_t^j for learner j
7          X^j ⟵ ⋃_{⟨i,j,0⟩∈ℰ} Y^i
8          # Carry out the training per se
9          ⟨non-trivial^j, θ^j⟩ ⟵ h^j(X^j)
10         # Dirty−up descendants if learner j is dirty
11         if non-trivial^j:
12             is-dirty^j ⟵ TRUE
13         if is-dirty^j:
14             for ⟨j, j̃, 0⟩ ∈ ℰ:
15                 is-dirty^j̃ ⟵ TRUE
16         # Finally compute outputs on training−set inputs
17         UseOnTrain(j, t, t_previous)
```

The UseOnTrain procedure is the heart of the compositional step. It allows the training set of a learner to be created from the results of training a predecessor j, by computing the outputs of learner j on each row of its training set. Two scenarios need to be considered:

(1) If the learner is marked "dirty" (resulting from non-trivial training or changes to the training set), the train-time state variables $\tilde{\xi}_0^j$ need to be reinitialized (line 4) in order to start computing the outputs from the first element of the training set. Then a sequence of output computations (f) and state transitions (g) are performed for each element of the training set, until the last time-step t (lines 5–7).

(2) If, on the other hand, the learner has remained "non-dirty" after its previous training, the training outputs can be computed incrementally from the previous results of UseOnTrain, where the starting time-step $t_{previous}$ is passed as an argument. This loop is very similar to the previous one, but does not involve reinitializing the train-time state variables (lines 9–11).

Listing 3. Training Output Computation

```
1  def UseOnTrain(j, t, t_previous):
2      # If learner is dirty, start a new trajectory
3      if is-dirty^j:
4          ξ̃_0^j ⟵ ξ_0^j
5          for τ in 0,...,t:
```

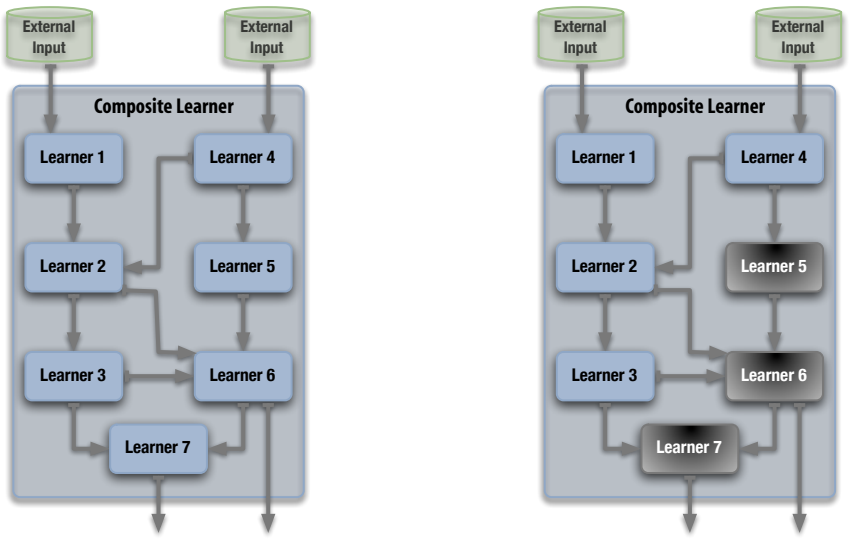


Fig. 8. **Left:** Example of a directed acyclic graph of composed learning modules. **Right:** Illustration of how the "dirty" status of a learner propagates to its descendants during training. As soon as a learner (here Learner 5) exhibits non-trivial retraining or its historical input data underwent revisions, its outputs on the training set must be recomputed anew, indicated by the darker shade. This applies recursively to all learners that directly or indirectly depend upon it.

$Y_\tau^j \longleftarrow f^j(X_\tau^j, \boldsymbol{\theta}^j, \tilde{\xi}_\tau^j)$
$\tilde{\xi}_{\tau+1}^j \longleftarrow g^j(X_\tau^j, Y_\tau^j, \boldsymbol{\theta}^j, \tilde{\xi}_\tau^j)$
else:
for τ **in** $t_{previous} + 1, \ldots, t$:
$Y_\tau^j \longleftarrow f^j(X_\tau^j, \boldsymbol{\theta}^j, \tilde{\xi}_\tau^j)$
$\tilde{\xi}_{\tau+1}^j \longleftarrow g^j(X_\tau^j, Y_\tau^j, \boldsymbol{\theta}^j, \tilde{\xi}_\tau^j)$

4.4 Output Computation

The ComputeOutput procedure traverses each learner j in the order given by the topological sort of the lag-0 edges in $\mathcal{E}$. For each, it constructs the inputs x_t^j for the learner from the outputs $y_{t'}^i$ of its "predecessors" in $\mathcal{E}$, whose existence the topological sort guarantees for lag-0 dependencies, and the top-level sequential validation loop likewise guarantees for lag-k dependencies, $k > 0$. As mentioned previously, we assume that an output y_t^i for $t < 0$ maps to a missing value default.

Listing 4. Output Computation

def ComputeOutput(t):
for j **in** $1, \ldots, J$:
 # Construct the input x_t^j for learner j
$x_t^j \longleftarrow \bigcup_{\langle i,j,\tau\rangle \in \mathcal{E}} y_{t-\tau}^j$
 # Compute output for learner j given input x_t^j

6 $y_t^j \longleftarrow f^j(x_t^j, \boldsymbol{\theta}^j, \hat{\xi}_t^j)$

Finally, the TransitionState procedure steps the state variables of each learner forward, given the current inputs and outputs.

Listing 5. State Transition

```
1 def TransitionState(t):
2     for j in 1,...,J:
3         ξ̂^j_{t+1} ⟵ g^j(x^j_t, y^j_t, θ^j, ξ̂^j_t)
```

4.5 Refinements

At the expense of some increased complexity, the practical efficiency of these algorithms can be improved (albeit not their asymptotic time complexity). In particular, the UseOnTrain procedure may benefit from the following improvement: a more sophisticated specification of the training function h would report on not only whether training has been non-trivial, but also from *where in the past* the newly-trained learner would want to revise its outputs. If a complete history of the training state $\tilde{\xi}_\tau^j$ is kept along the training set, this allows us to avoid resetting the state to zero when a "non-trivial training" has occurred (line 4 of UseOnTrain); instead we can only go back to the time of the furthest revision.

A lesser benefit can be had at no space cost in the case of history revisions occurring at the last time-step of a given training set. It comes from the recognition that the very last state-transition in the UseOnTrain loops of lines 5–7 and 9–11 (for $\tau = t$) can be delayed until the start of the next UseOnTrain call without changing behavior. (A casual perusal of this function reveals this transformation leaves unchanged the overall ordering of the calls to the underlying f^j and g^j functions.) Delaying this state-transition opens up the option of *entirely omitting the transition* should it be found unnecessary. Consider, within a current call to UseOnTrain, a history revision that is anticipated to occur in the very last time-step of the training set (currently at time t, which would become time t_{previous} in the next call to UseOnTrain). If such a revision indeed occurs, UseOnTrain can handle it by starting the loop of line 9 at time t_{previous} (instead of $t_{\mathit{previous}} + 1$), *overwriting the previously-computed outputs with those computed from the revised inputs.* If no revision occurs, UseOnTrain simply computes the missing state-transition for $\tau = \mathit{previous}$ before starting loop 9–11.

Although this special case seems rather far-fetched, we remark that it corresponds to the real-world situation of a simulation with monthly data, with the common case of macro-economic series subject to last-month revisions.

We briefly mention two further obvious improvements: since UseOnTrain is only required to be called on a learner to prepare the training set for its children in $\mathcal{E}$, this call is evidently not required for leaf learners, and may be omitted. In addition, the overall partitioning of the learning problem along a graph structure naturally lends itself to a parallel implementation for regions of the graph that show no dependencies.

5. DOMAIN ANALYSIS

The definition of temporal learning algorithm introduced in §3.1/p. 10 lends itself will to composition through network structures. In this section we present a short domain analysis to better understand why these structures are necessary over the arguably simpler hierarchical (tree-like) alternatives for a number of useful machine learning problems.

5.1 Hierarchical Designs

In a hierarchical composition of "learning modules", a single learner owns a set of "sublearners" which can be used for performing auxiliary work. The *training* and *output computation* functions of a learner are trivially defined as first recursively calling the corresponding function of each sublearner, and then carrying out any required additional work. As to the state representation in the case of dynamic models, two possibilities arise:

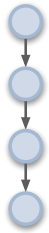
—Each learner can *contain* its own state[25], which is directly controlled (e.g. reset to an initial starting point) by an interface provided by the learner. However, this type of interface, by almost completely abstracting away the state representation, makes it unduly difficult to operate the learner along multiple parallel trajectories. In a compositional context, this is always necessary in order to compute the fitted values on the training set in order to construct the input needed by a subsequent learner. In a simulation context, Monte Carlo trials require a large number of separate states.

—Each learner can be passed a state vector at output-computation time; this vector contains both the state required by the learner itself, but also the concatenation of state vectors required by sublearners. This arrangement poses two difficulties: first it requires the state of the sublearners to be of a known fixed size; second, and this is a general problem with hierarchical composition, it does not allow the work done by a learner to be shared by others. This second situation is detailed next.

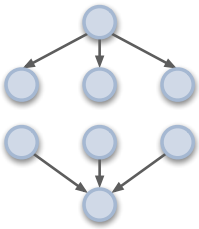
5.2 Network Designs

A general problem with an hierarchical decomposition of learners is that it is impossible to factor out computations that would be required by two separate learners. This operation requires a more general graph representation. Directed acyclic graphs are sufficient for a large number of causal time-series forecasting and temporal decision-making tasks, as the following cases illustrate:

► Chain Graphs : Chain graphs correspond to a normal processing pipeline, starting with input variable preprocessing (including, for instance, dimentionality reduction), followed by a traditional learning stage, and postprocessing. They also encompass the case of *stacking* [Wolpert 1992], wherein a cascade of learners correct previous ones' errors.

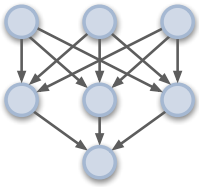


[25] A *has-a* relationship in object-oriented parlance; see, e.g., [2007].



▶ Fan Out : This corresponds to factored-out computations, such as the use of a common covariance matrix in otherwise-parallel tasks.

▶ Fan In : This corresponds to the combination of several previously-computed results; in machine learning, classic examples are mixture of experts [Jordan and Jacobs 1994] and bagging [Breiman 1994]. In a financial setting, one can view the mean-variance optimization step of a portfolio selection problem [Markowitz 1959] as a graph of this type, wherein expected returns and covariances are first estimated (along with additional inputs such as market equilibrium weights and investment views if one is following a more robust methodology such as Black-Litterman allocation [Black and Litterman 1992; Fabozzi et al. 2007]), followed by their combination in the optimization step *per se*.



▶ Lattices : More complex lattices arise naturally in the context of *hyper-mixtures*. These are a generalization of mixture models that occur when one is performing sequential hyperparameter optimization. Consider an exponentiated gradient mixture (Herbster and Warmuth 1998; financial applications are considered in Helmbold et al. 1998) that combines, through weighted averaging, the outputs of a set of base models (which are represented as the top layer in the illustration). At time-step t, the mixture associates a weight $w_{t,i}$ to model i. The weights are adapted at each time-step according to an error criterion aiming at putting more importance on the recently better-performing models. This adaptation is subject to a set of hyperparameters Θ controlling, for instance, the adaptation speed and the maximum weight that can be put on a single model. The problem is that one does not know *a priori* what are good settings for these hyperparameters. In the spirit of sequential validation, a solution is to run a number of such mixtures in parallel (depicted as the middle layer in the illustration), each with its own hyperparameters Θ_j, and have the hyper-mixture (bottom node in the illustration) select the best one on the basis of the past performance of each mixture.[26]

In order to avoid repeated computation, it is desirable to let the base models be *shared* among the mixtures, each one then performing its own independent weighting. At time t, this yields the following sequence of computations:

(1) The set of base models $\{i\}$ compute their outputs.
(2) The individual mixtures, each with adaptation hyperparameters Θ_j, combine these outputs according to weights $w^j_{t,i}$; the mixtures independently update their weights.
(3) The hyper-mixture selects the best mixture at time t according to the recent performance of each mixture.

[26] One may argue that this does not completely solve the problem since the time horizon over which the performance is evaluated at the hyper-mixture level is itself an hyperparameter; in practice, this hyperparameter is much easier to choose "reasonably" according to the characteristics of the problem than those at the mixture level.

It should be emphasized that time-delays (i.e. using as input a result that was computed in a previous time-step) do not introduce an ordering dependency in the directed acyclic graph; they simply correspond to buffering operations that must be carried out during propagation.

5.2.1 *State Representation.* In §5.1/p. 21 we examined why state containment and simple aggregation are inappropriate for a variety of sensible use-cases. The solution to state representation is to designate a *state-holder object* which associates a learner reference to an arbitrary state for the learner; each learner uses the state-holder object to find its own state. It is the state-holder object, and not individual state objects, that is communicated among learners within the compositional steps of §4/p. 15.

5.3 Why Separate Training and Output Computation?

A sequential learning setting is often associated with *online learning* [Saad 1999], in which model parameters are adapted contemporaneously with the computation of outputs, as examples are presented to the learner. As such, it can be asked why the framework introduced in §3/p. 10 distinguishes between training and output computation. Our prototype example for answering this question is to consider the Kalman Filter considered in §3.1.1/p. 12, which has both parameters and dynamic state. The output function determines the model output given its state, and the state-transition function determines the next state from the current one. The training procedure corresponds to the estimation of parameters necessary for both the output and state-transition functions.

In this framework, training operations correspond to *batch training*: given a complete past history, we can use it all (without cheating or necessarily taking temporality into account) to update the model parameters. These should be considered offline (time-invariant) parameters, independent of the current state. Furthermore, the training set needs not correspond to a single time history; several histories (either generated by a random process, or corresponding to several parallel histories of real data, e.g. many historical stock price histories) may be combined into a single training set, with the resulting maximum likelihood estimation remaining well defined.

In contrast, in the traditional online learning context, parameters are updated at each time-step within a trajectory as a new example is received. This is easily accomodated in the framework of §3/p. 10 by considering the (online) parameters to be part of the state vector. In this context the state transition equation may be viewed as forming the learning step. Batch training is simply omitted in this case.

6. CASE STUDY

In this final section, we review the implementation of a complex financial decision-making system making use of the composition primitives introduced previously. This system is designed to automate the trading a portfolio of commodity futures (see Hull [2005] for an introduction), and relies on a number of learning algorithms at various points in the process of turning raw data into actionable portfolion recommendations.

6.1 Existing Systems

The majority of Commodity Trading Advisors (CTAs) and automated trading systems use active methodologies, of which trend following in one fashion or another is especially prevalent [Fung and Hsieh 2001; Spurgin 1999]; these methods rely on the postulate that past price movements are good predictors of future ones. This is in part motivated by theoretical findings that document the evidence of abnormal returns from momentum strategies in commodities after adjusting for systematic and time-varying risks [Erb and Harvey 2006; Miffre and Rallis 2007], echoing the classical results of Jegadeesh and Titman [1993] for equities.

Unfortunately, many of the systems deployed in practice suffer from *retrospective bias* to some extent, namely that the parameters governing the trading rules are adjusted *ex post* to yield good past performance, without any guarantees other than a trader's "gut feeling" that they would perform acceptably in the future.

The design of a "cheating-free" trading system can only be realized by reproducing the real-time actions of a decision-maker, making sequential validation a natural framework. This system has been developed and used in an industrial context.

6.2 Making Use of the Composition Primitives

Figure 9 illustrate the overall system, decomposed into relevant functional blocks. We see that it consists into two "core models" (at the top), a technical model mostly making use of recent asset price variations, and a fundamental model, which can exploit more complex relationships between assets and is used to construct a mean-variance efficient portfolio. The rest of the system (various mixture models) is used to combine the two basic building blocks. As will be made clear below, the apparent complexity of this combination arises because we want to set as few hyperparameters as possible and let the system adaptively choose the best hyperparameter settings.

▶ A. Technical Model : The first "core" building-block, illustrated in Fig. 10, is a traditional moving-average based momentum model that takes long and short positions based on an assetwise comparison between the current price and the one-year price moving average.[27] We introduce a distinction between portfolios of long and short positions, based on the observation that momentum tends to be more persistent for shorts than longs [Miffre and Rallis 2007]. More specifically, let p_{it} the price of asset i at time t and $\hat{p}_{it}$ the one-year moving average of the same quantity. The "long-side momentum" rule is simply to recommend a long position if $p_{it} > \hat{p}_{it}$ and a neutral position otherwise; conversely, the "short-side momentum" rule recommends a short position if $p_{it} < \hat{p}_{it}$, and a neutral position otherwise. After the momentum recommendation is made, it is filtered according to the volatility rule of Dunis and Miao [2006]; this rule sets to neutral the momentum model positions in periods of high market volatility. In terms of the learning primitives of §3.1/p. 10, each learner in the technical model consists of fixed decision rules with online estimation of key thresholds (current price moving average and volatility). As such, they do not make use of the batch training facilities.

[27] "Long" means a buying position in the asset, "short" means a selling position, and "neutral" means that no position is taken. Commodity futures can as easily be sold short as they can be bought, without the restrictions that affect equities.

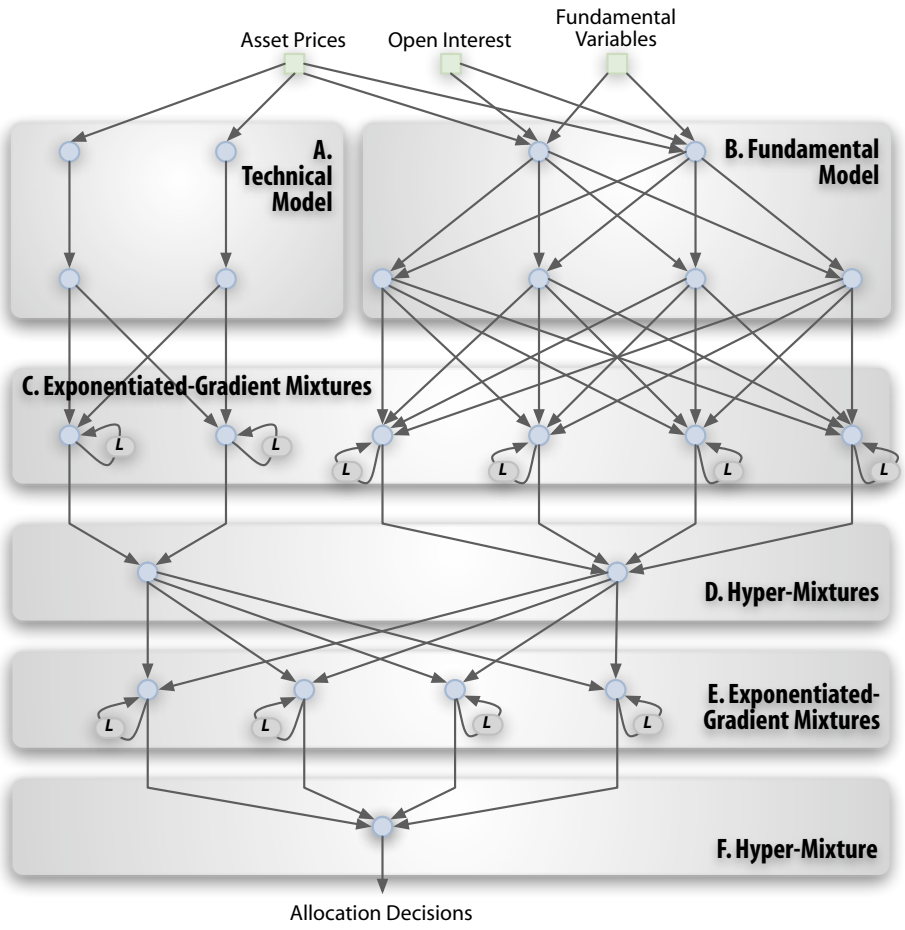


Fig. 9. Functional blocks of a financial portfolio allocation system making use of the composition primitives introduced previously. The square (green) vertices represent external inputs. The round (blue) vertices represent temporal learning algorithms in the sense of definition 1. The loops labeled with an L represent time-lagged connections.

It remains to combine the resulting independent long and short portfolios; this is performed adaptively using a mixture described below.

▶ B. Fundamental Model : The fundamental model seeks to exploit deeper relationships between assets, trying to estimate asset expected returns and covariances over the next month and constructing a mean-variance efficient portfolio [Markowitz 1959] to exploit the relationships that have been found. Unusual for commodity trading systems, we rely on the Black and Litterman [1992] methodology of portfolio optimization as a second building-block (see Fig. 11). This makes use of an asset expected-return predictor, which uses locally-weighted linear ridge regression [Hoerl and Kennard 1970] from a number of technical and macroeconomic input variables, with hyperparameters—such as the ridge coefficient and the number of

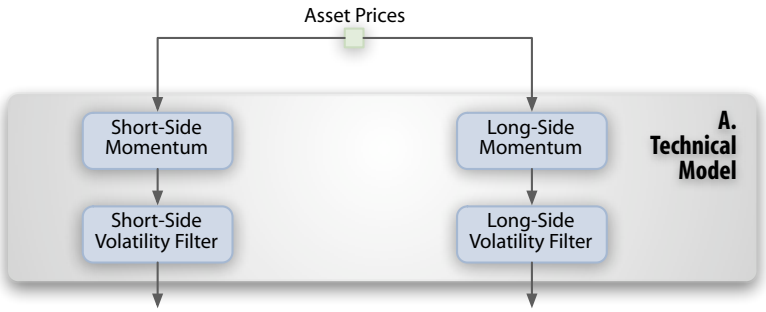


Fig. 10. Details of the technical model.

neighbors in the local window—optimized by a grid search at each time-step over a validation set.

We also require an asset-return covariance matrix estimator, an exponentially-weighted moving average following the RiskMetrics [1996] methodology being used for simplicity. This carries out a recursive (online) update of the current covariance estimator $\hat{\mathbf{\Sigma}}_t$ given the previous estimator and the vector of time-t asset returns $\mathbf{r}_t$,

$$\hat{\mathbf{\Sigma}}_t = \lambda\, \hat{\mathbf{\Sigma}}_{t-1} + (1-\lambda)\mathbf{r}_t\mathbf{r}_t',$$

where the decay factor $\lambda = 0.94$ was used, consistently with the RiskMetrics recommendations for daily data.

The Black-Litterman model also requires the market capitalization of each asset in order to determine the asset equilibrium returns, which are part of the procedure. Since commodities do not have a capitalization in the traditional sense, we relied on the value of the outstanding open interest as a capitalization proxy.

Black-Litterman (B-L) finds the mean-variance-efficient portfolio weights, namely the result of the optimization problem

$$\mathbf{w}_t^* = \arg\max_{\mathbf{w}} \mathbf{w}'\boldsymbol{\mu}_t^{BL} - \frac{\gamma}{2}\mathbf{w}'\hat{\mathbf{\Sigma}}_t\mathbf{w},$$

where $\boldsymbol{\mu}_t^{BL}$ is the vector of expected asset returns obtained by the Black-Litterman estimator (which uses the raw asset return estimator; see Black and Litterman [1992] for details), and γ is the investor risk-aversion coefficient. The B-L allocation model relies on a number of such hyper-parameters. Since we do not know *a priori* what should be good values for them, we run a number of B-L allocations in parallel (denoted "Black-Litterman $1 \ldots 4$" on the figure), and mix them according to a mixture.

In terms of learning primitives, of the components depicted in Fig. 11, the asset-return forecast contains a full-scale training operation and does not need to carry trajectory-specific state. In contrast, the covariance model uses online estimation, and hence can dispense with the necessity for batch training. The Black-Litterman model neither needs training nor specific state: the portfolio optimization is performed on-the-fly within the output-computation operation.

▶ C. Exponentiated-Gradient Mixtures : Despite their differences, a common

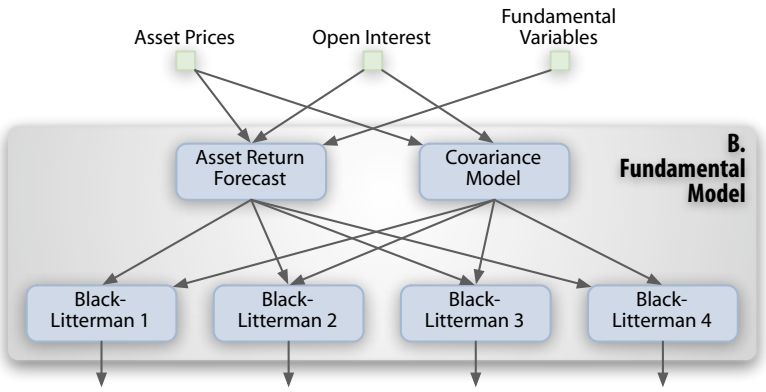


Fig. 11. Details of the fundamental model.

thread to both the technical and fundamental models is that they *output several parallel hypotheses*, corresponding to different choices of hyperparameters (the mixing proportions of the long and short models in the cast of the technical model, and the B-L hyperparameters in the case of the fundamental model). Instead of carrying out hard model selection, we combine these hypotheses with the help of a mixture, using the fixed-share version [Herbster and Warmuth 1998] of the exponentiated gradient algorithm [Kivinen and Warmuth 1997]. This method uses a multiplicative update of the weights, followed by a redistribution step that prevents any of the weights from becoming too large. Briefly, this mixture operates as follows. Let $C_{t,k}$ a performance measure observed for model k (out of a total of K models) during period t. The weight given to each model is updated according to:

(1) *Performance Update:*

$$w^m_{t,k} = w^s_{t,k} \exp(\eta\, C_{t,k}),$$

(2) *Fixed Share:*

$$\begin{aligned} \text{pool}_t &= \sum_{k=1}^{K} w^m_{t,k} \\ w^s_{t+1,k} &= (1-\alpha) w^m_{t,k} + \frac{\text{pool}_t - \alpha w^m_{t,k}}{K-1}, \end{aligned}$$

(3) *Final Weight Normalization* (the weight attributed to each model by the mixture is given by $v_{t+1,k}$):

$$v_{t+1,k} = \frac{w^s_{t+1,k}}{\sum_j w^s_{t+1,j}}.$$

In these equations, η is a constant *learning rate* hyperparameter that controls the responsiveness of the weight adaptation. The α hyperparameter, along with the intermediate variable pool_t, define a "sharing mechanism" whereby more important models distribute a part of their weight to the less important models; this is important to prevent some models from having their weight decay to zero and enables

the mixture to better track nonstationarities. Obviously, it remains to choose how to fix those parameters. In the spirit of "letting the data speak for itself", we run several of those mixtures in parallel, varying the η parameter between a "fast" and a "slow" mixture, and make the final determination by the following step.

An extensive analysis of the exponentiated gradient mixture, including bounds on the generalization error, is provided by Herbster and Warmuth [1998].

In terms of learning primitives, the mixture operates entirely in an online fashion, and requires no separate batch training step.

▶ D. Hyper-Mixtures : As noted, in the exponentiated-gradient algorithm, hyperparameters control the convergence rate and the minimum value of a weight; these are not known *a priori*, so we run several such mixtures in parallel. The hyper-mixture step *selects* (rather than combines) the best-performing mixture over a one-year horizon. The selection is based on a financial performance criterion (the Sharpe [1966] ratio, which measures the average excess return over the risk-free rate per unit of standard deviation of portfolio returns). Note that two separate selections are carried out, one for the technical and one for the fundamental model.

▶ E. Exponentiated-Gradient Mixture : Finally we must combine the technical and fundamental sides into a final single allocation. Again, since mixing weights are not known *a priori* another layer of mixtures (several of them since their hyperparameters are not known either) is used to form an adaptive combination.

▶ F. Final Hyper-Mixture : This final layer selects the best-performing mixture in the same spirit as step D, also on a one-year horizon, and considering a financial performance criterion.

In Figure 9, each vertex (except for the input variables) is considered a temporal learning algorithm in the sense of definition 1, even though the full ramifications of the definition are not necessary in all instances. (For instance the mixtures only perform online adaptation and do not require a training part.) This allows the algorithms of section 4 to be applied with no modification, resulting in a flexible and robust overall system. Figure 12 illustrates the simulated financial performance of the strategy embedded by the composite learner strategy.

7. CONCLUSION

It is obvious that the functional composition of models is as old as statistical modeling itself. In the machine-learning literature, a number of composition mechanisms and "meta-algorithms" have been proposed, but much less attention has been paid to the engineering issues surrounding composition.

In parallel to this, machine-learning research overwhelmingly employs held-out tests to evaluate the out-of-sample performance of models and as a general framework allowing unbiased comparison between models. These approaches have been becoming more popular among finance researchers in recent years as well. Excepting technical details such as those illustrated in Figure 4, out-of-sample testing provides a true unbiased estimator of future performance (under the strong assumption that the data-generating process is IID). For data in which ordering matters, *sequential validation* can be applied as a reasonable alternative.

We examined the issues surrounding composition in the presence of sequential

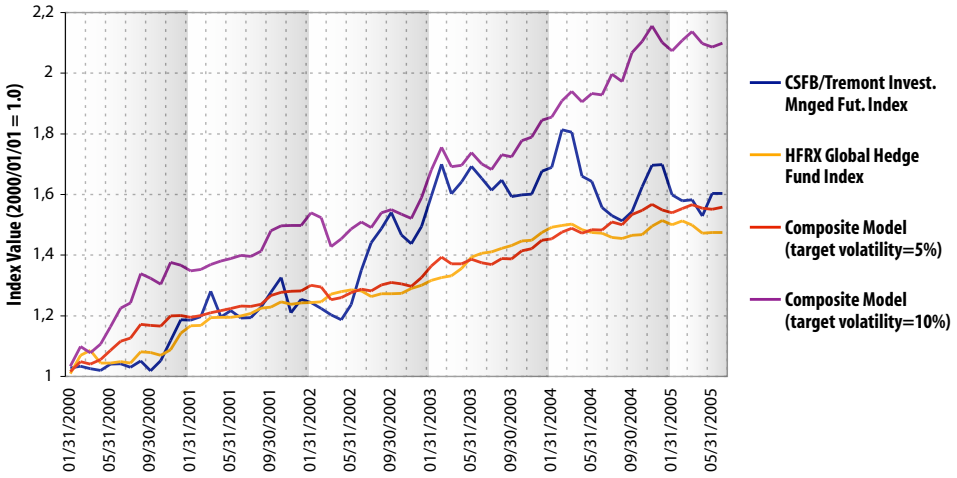


Fig. 12. Cumulative return of the composite-learner-based strategy over the 2000–05 period, for two levels of annual target volatility (5% and 10%). For comparison purposes, the performance of two well-known hedge fund indexes are illustrated.

validation. After introducing the challenges inherent in the sequential validation evaluation of learning algorithm performance, including the correct handling of revisions in economic time series, we introduced a definition of temporal learning that lends itself very well to a set of systematic algorithms to efficiently and correctly implement sequential validation in practical contexts. These algorithms take care of keeping each training set, and by extension the learners in a temporal learning network, up-to-date with respect to its sources; as such, they free the engineer to concentrate on the true difficulties of the problem s/he is facing, and not the incidental details of composing several models into a working system.

We illustrated the flexibility of the approach by intorducing a portfolio allocation system made up of a number of learners, from the fixed decision rules, to batch learners, to online learners, and how they can be naturally combined to form a powerful decision-support system whose performance can be rigorously ascertained.

REFERENCES

Abu-Mostafa, Y. S., Atiya, A. F., Magdon-Ismail, M., and White, H. 2001. Special issue on neural networks in financial engineering. *IEEE Transactions on Neural Networks 12,* 4 (July), 653–656.

Bishop, C. M. 2006. *Pattern Recognition and Machine Learning*. Springer, New York, NY.

Black, F. and Litterman, R. 1992. Global portfolio optimization. *Financial Analysts Journal 48,* 5 (September), 28–43.

Bollerslev, T. 1986. Generalized autoregressive conditional heteroskedasticity. *Journal of Econometrics 31*, 307–327.

Booch, G., Maksimchuk, R. A., Engel, M. W., Young, B. J., Conallen, J., and Houston, K. A. 2007. *Object-Oriented Analysis and Design with Applications*, Third ed. Addison-Wesley, Reading, MA, USA.

Breiman, L. 1994. Bagging predictors. *Machine Learning 24,* 2, 123–140.

Campbell, J. Y., Lo, A. W., and MacKinlay, A. C. 1997. *The Econometrics of Financial Markets*. Princeton University Press, Princeton, NJ, USA.

Chapados, N. and Bengio, Y. 2001. Cost functions and model combination for VaR-based asset allocation using neural networks. *IEEE Transactions on Neural Networks 12,* 4 (July), 890–906.

Clements, M. P. and Hendry, D. F. 1999. *Forecasting Non-stationary Economic Time Series*. Zeuthen Lecture Book Series. MIT Press, Cambridge, MA.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. 2001. *Introduction to Algorithms*, Second ed. MIT Press, Cambridge, MA.

Dempster, A. P., Laird, N. M., and Rubin, D. B. 1977. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society, B 39,* 1, 1–38.

Dunis, C. and Miao, J. 2006. Volatility filters for asset management: An application to managed futures. *Journal of Asset Management 7*, 179–189.

Engle, R. 1982. Autoregressive conditional heteroscedasticity, with estimates of the variance of united kingdom inflation. *Econometrica 50*, 987–1007.

Erb, C. B. and Harvey, C. R. 2006. The strategic and tactical value of commodity futures. *Financial Analysts Journal 62,* 2 (March/April), 69–97.

Fabozzi, F. J., Kolm, P. N., Pachamanova, D. A., and Focardi, S. M. 2007. *Robust Portfolio Optimization and Management*. The Frank J. Fabozzi Series. John Wiley & Sons, Hoboken, NJ.

Fama, E. F. and French, K. R. 2002. The equity premium. *The Journal of Finance 57,* 2, 637–659.

Freund, Y. and Schapire, R. E. 1996. Experiments with a new boosting algorithm. In *Machine Learning: Proceedings of Thirteenth International Conference*. ACM, USA, 148–156.

Fung, W. and Hsieh, D. A. 2001. The risk in hedge fund strategies: Theory and evidence from trend followers. *Review of Financial Studies 14,* 2, 313–341.

Ghahramani, Z. and Hinton, G. E. 1996. Parameter estimation for linear dynamical systems. Tech. Rep. CRG-TR-96-2, Department of Computer Science, University of Toronto.

Gingras, F., Bengio, Y., and Nadeau, C. 2002. On out-of-sample statistics for time-series. Tech. Rep. 2002s-51, CIRANO. Available at `http://www.cirano.qc.ca/pdf/publication/2002s-51.pdf`.

Greene, W. H. 2002. *Econometric Analysis*, Fifth ed. Prentice Hall, Englewood Cliffs, NJ.

Hansen, P. R. and Lunde, A. 2005. A forecast comparison of volatility models: does anything beat a garch(1,1)? *Journal of Applied Econometrics 20,* 7, 873–889.

Hastie, T., Tibshirani, R., and Friedman, J. 2001. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, Berlin, New York.

Hautsch, N. and Hess, D. 2002. The processing of non-anticipated information in financial markets: Analyzing the impact of surprises in the employment report. *European Finance Review 6,* 2 (January), 133–161.

Helmbold, D. P., Schapire, R. E., Singer, Y., and Warmuth, M. K. 1998. On-line portfolio selection using multiplicative updates. *Mathematical Finance 8,* 4, 325–347.

Herbster, M. and Warmuth, M. K. 1998. Tracking the best expert. *Machine Learning 32,* 2, 151–178.

Hess, D. 2001. Surprises in u.s. macroeconomic releases: Determinants of their relative impact on t-bond futures. Tech. rep., Center of Finance and Econometrics, University of Konstanz. Discussion Paper No. 2001/01. Available at SSRN: `http://ssrn.com/abstract=267492`.

Hoerl, A. and Kennard, R. 1970. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics 12,* 3, 55–67.

Hull, J. C. 2005. *Options, Futures and Other Derivatives*, Sixth ed. Prentice Hall, Englewood Cliffs, NJ.

Jegadeesh, N. and Titman, S. 1993. Returns to buying winners and selling losers: Implications for stock market efficiency. *Journal of Finance 48,* 1 (March), 65–91.

Jordan, M. I. and Jacobs, R. A. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural Computation 6*, 181–214.

Kailath, T., Sayed, A. H., and Hassibi, B. 2000. *Linear Estimation*. Prentice Hall, Englewood Cliffs, NJ.

Kalman, R. E. 1960. A new approach to linear filtering and prediction problems. *Transactions of the American Society of Mechanical Engineering, Series D, Journal of Basic Engineering 82*, 34–45.

Kivinen, J. and Warmuth, M. K. 1997. Exponentiated gradient versus gradient descent for linear predictors. *Information and Computation 132,* 1, 1–63.

Lo, A. W. and MacKinlay, A. C. 1990. Data-snooping biases in tests of financial asset pricing models. *Review of Financial Studies 3,* 3, 431–467.

Markowitz, H. M. 1959. *Portfolio Selection: Efficient Diversification of Investment*. John Wiley & Sons, New York, London, Sydney.

Miffre, J. and Rallis, G. 2007. Momentum strategies in commodity futures markets. Tech. rep., EDHEC Risk and Asset Management Research Centre, Nice, France. Forthcoming in Journal of Banking and Finance.

Neal, R. M. and Hinton, G. E. 1998. A view of the EM algorithm that justifies incremental, sparse and other variants. In *Learning in Graphical Models*, M. I. Jordan, Ed. MIT Press, Cambridge, MA, 355–368.

RiskMetrics. 1996. Riskmetrics—technical document. Tech. rep., J.P. Morgan, New York, NY. `http://www.riskmetrics.com`.

Roweis, S. and Ghahramani, Z. 1999. A unifying review of linear gaussian models. *Neural Computation 11*, 305–345.

Saad, D. 1999. *On-Line Learning in Neural Networks*. Cambridge University Press, Cambridge, UK.

Shadbolt, J. and Taylor, J. G. 2002. *Neural Networks and the Financial Markets: Predicting, Combining and Portfolio Optimisation*. Springer, Berlin, New York.

Sharpe, W. F. 1966. Mutual fund performance. *Journal of Business 39,* 1 (January), 119–138.

Shumway, R. H. and Stoffer, D. S. 1982. An approach to time series smoothing and forecasting using the em algorithm. *Journal of Time Series Analysis 3,* 4, 253–264.

Spurgin, R. 1999. A benchmark for commodity trading advisor performance. *Journal of Alternative Investments 2,* 1, 11–21.

Stock, J. H. and Watson, M. W. 1996. Evidence on structural instability in macroeconomic time series relations. *Journal of Business and Economic Statistics 14*, 11–30.

Stock, J. H. and Watson, M. W. 1999. Forecasting inflation. *Journal of Monetary Economics 44*, 293–335.

Stone, M. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society, B 36,* 1, 111–147.

White, H. 2000. A reality check for data snooping. *Econometrica 68,* 5, 1097–1126.

Wolpert, D. 1992. Stacked generalization. *Neural Networks 5*, 241–249.

Received Oct. 2008;